The Def-inite Approach to Dependency Analysis

Samir Genaim and Michael Codish
The Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
{genaim,mcodish}@cs.bgu.ac.il

Abstract

We propose a new representation for the domain of Definite Boolean functions. The key idea is to view the set of models of a Boolean function as an incidence relation between variables and models. This enables two dual representations: the usual one, in terms of models, specifying which variables they contain; and the other in terms of variables, specifying which models contain them. We adopt the dual representation which provides a clean theoretical basis for the definition of efficient operations on Def in terms of classic ACI1 unification theory. Our approach illustrates in an interesting way the relation of Def to the well-known set-Sharing domain which can also be represented in terms of sets of models and ACI1 unification. From the practical side, a prototype implementation provides promising results which indicate that this representation supports efficient groundness analysis using Def formula.

1 Introduction

Boolean functions play an important role in various formal methods for specification, verification and analysis of software systems. In program analysis, Boolean functions are often used to approximate properties of the set of states encountered at a given program point. For example, a conjunction $x \wedge y$ could specify that variables x and y satisfy some property whenever control reaches a given program point. A Boolean function $\varphi_1 \rightarrow \varphi_2$ could specify that if φ_1 is satisfied at a program point (perhaps depending on the unknown inputs to the program) then also φ_2 is satisfied. A disjunction $\varphi_1 \vee \varphi_2$ could arise as a consequence of a branch in the control where φ_1 and φ_2 approximate properties of the `then` and `else` branches respectively.

For program analysis, we often consider the *positive* Boolean functions, Pos. Namely, those for which $f(true, \ldots, true) = true$. This restriction is natural as, due to the element of approximation, the result of an analysis is not a "*yes/no*" answer, but rather a "*yes/maybe not*" answer. In this case there is no "negative" information. Sophisticated Pos-based analyzers implemented using binary decision diagrams (BDD's) [4] have been shown [12] to give good experimental results both with regards to precision as well as with regards for the efficiency of the analyzers.

However, scalability is a problem and inputs (programs) for which the analysis requires an exponential number of iterations or exponentially large data structures are encountered [6].

In general, scalability problems with program analyses are often tackled by introducing widening operations [13]. Widening techniques can be applied to restrict the number of iterations as well as to limit the size of data-structures. A disadvantage of the use of binary decision diagrams for Pos-based groundness analyses is the lack of suitable widening techniques which maintain reasonable precision on the one hand and guarantee scalability on the other.

The domain, Def, of definite Boolean functions is a subdomain of Pos. These are the positive functions whose sets of models are closed under intersection. The domain Def is less expressive than Pos. For example, the formula $x \vee y$ is not in Def. However, Def-based analyzers can be implemented using less complex data structures and can be faster than Pos-based analyzers. As for precision, Def has been shown to provide a reasonable tradeoff between efficiency and precision for goal dependent groundness analyses (where a description of the inputs to the program being analyzed is given) [19, 18]. Our present work indicates that in practice, for a large set of benchmark programs, goal independent Def analysis is just as precise as goal dependent analysis with respect to the groundness information derived. However, scalability remains a problem for Def. In [16], the authors demonstrate a series of inputs (programs) for which Def-based groundness analysis involves $2^{\sqrt{n}}$ iterations (where n is the size of the input). An advantage of basing analyses on Def is the fact that widening operations are not difficult to define and implement.

This paper investigates a new representation for Def functions. The key idea is to view a set of sets of variables (the models of a Boolean function) as an incidence relation between variables and sets. This leads to two dual views of the structure. The usual view looks at the sets and specifies which are their elements. The dual view looks at the variables and specifies which are their sets. We take the dual view and show that Def is isomorphic to the domain of tuples of sets of variables modulo ACI1 equivalence. The approach and the design of our domain build on two previous results, presented in [8] and in [9, 10], which focus on two types of analysis for logic programs that have received considerable attention. The first is the analysis of possible *set-sharing*, that is, sharing amongst sets of program variables using the Sharing domain introduced by Jacobs and Langen. The second is the analysis of definite groundness dependencies using domains such as Def and Pos as suggested by Dart [14] and by Marriott and Søndergaard [20].

The work presented in [8] illustrates the application of logic programming with sets (of variables) to support an isomorphic representation for set sharing analysis. This approach is theoretically pleasing as the operations for sharing analysis are based on a classic ACI1 equality theory.

In [9, 10], the authors present an isomorphism between Sharing and Pos which leads to a better understanding of the similarities between groundness and sharing analyses. In particular Def analyses in the two domains are shown to coincide exactly extending an important result by Cortesi *et al.* [11]. From the practical point of view, the consequence is that the operations in a Sharing analysis are simplified when the focus is restricted to Def information.

This result is applied by King *et al.* in [19] where the authors quotient the Sharing domain to perform groundness analysis in Def. Their analyzer is reported to perform well in comparison to a Pos analysis using BDD's. In a sequel to [19] described in [18], the authors obtain a much faster analysis by factoring the representation into three components: G (ground arguments), E (equivalent arguments) and P (a propositional Horn clause component). The G and the E components can be represented as an atom containing only variables and the constant $true$. For example, the atom $p(A, true, A, B, true)$ represents the Def function $(x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$ on five variables. The approach is similar to the GER representation for Pos described in [3].

This paper, supports the view that groundness dependencies in Def can be obtained efficiently by quotienting a Sharing analysis. We are motivated by the preliminary results described in [19] and challenged by the speed of the analyzer described in [18]. We argue that when adopting the set logic programming approach of [8] we obtain "for free" a factoring of the domain to consider seperately ground arguments and equivalence between arguments. Moreover, we obtain a representation which is easy to implement and to widen.

From the technical point of view, the set logic programming approach is attractive because we prove that the quotient of Sharing for Def in our representation is the well-studied and classic notion of ACI1 equivalence. Namely, two atoms with sets of variables as terms, representing corresponding Sharing elements are equivalent in Def if and only if they are ACI1 equivalent. All of the operations for Def analysis in Sharing are shown to be classic operations for ACI1 terms which are in our special case known to be efficient to maintain. This is in contrast to the use of ACI1 theory in the context of sharing analysis which requires a specialized notion of ACI1 unification (which chooses the most general unifier according to a non-standard ordering).

So, the quotient of Sharing for Def in our representation sums up to applying the standard ACI1 ordering on tuples of sets instead of the specialised ordering used in [8].

2 A Motivating Example

Table 1 illustrates various representations of positive Boolean functions which are relevant to our exposition. The first and second columns depict a positive Boolean function φ on n variables V and its set of models $[\![\varphi]\!]$. The set $\{x, y\}$ is written xy for short and $\perp$ denotes the empty set. Note that a positive function always contains V as one of its models. Throughout the example we take $n = 2$. In the third column the models of φ are pointwise complemented with respect to V. These are the models of the "dual negation" of φ which is denoted $coneg(\varphi)$. These are elements of the Sharing domain (which always contain $\perp$). Dual negation is discussed in [10] and shown to provide an isomorphism between Sharing and Pos. The forth column contains an n-tuple of sets ("Tuple_n") of variables disjoint from V. To obtain this representation, the models of $coneg(\varphi)$ are first given arbitrary names $(a, b, c, \ldots)$. The i^{th} set in a tuple then specifies which models of $coneg(\varphi)$ contain the i^{th} variable of V. For example, if we name the models in

$$[\![coneg(x \rightarrow y)]\!] = \{xy, x, \bot\}$$

by a, b and c respectively, then the Tuple_n representation is $\langle ab, a\rangle$: ab in the first position because x occurs in both a and b; and a in the second position because y occurs only in a. This is the representation used in [8] which is shown to be isomorphic to the Sharing domain when viewed modulo an appropriate notion of equivalence. It is also the representation we work with in this paper where we make the observation that looking at the elements of this column modulo standard ACI1 equivalence gives Def.

The fifth column in Table 1 illustrates how tuples of sets represent the strongest logical consequence $\varphi{\downarrow}$ of φ which is in Def. All of the functions in the example, except for $x \vee y$, satisfy $\varphi{\downarrow} = \varphi$ because they are already in Def. For $x \vee y$ we have $(x \vee y){\downarrow} = true$. In the sequel we will observe that the tuples $\langle a, b\rangle$ and $\langle ac, ab\rangle$ corresponding to $x \vee y$ and $true$ are ACI1 equivalent.

φ	$[\![\varphi]\!]$	$[\![coneg(\varphi)]\!]$	Tuple_n	$\varphi{\downarrow}$
$x \wedge y$	$\{xy\}$	$\{\bot\}$	$\langle \bot, \bot\rangle$	$(x \leftrightarrow true) \wedge (y \leftrightarrow true)$
x	$\{x, xy\}$	$\{y, \bot\}$	$\langle \bot, a\rangle$	$\exists_a.((x \leftrightarrow true) \wedge (y \leftrightarrow a))$
$x \leftrightarrow y$	$\{\bot, xy\}$	$\{xy, \bot\}$	$\langle a, a\rangle$	$\exists_a.((x \leftrightarrow a) \wedge (y \leftrightarrow a))$
$x \rightarrow y$	$\{\bot, y, xy\}$	$\{xy, x, \bot\}$	$\langle ab, a\rangle$	$\exists_{a,b}.((x \leftrightarrow a \wedge b) \wedge (y \leftrightarrow a))$
$x \vee y$	$\{x, y, xy\}$	$\{y, x, \bot\}$	$\langle a, b\rangle$	$\exists_{a,b}.((x \leftrightarrow b) \wedge (y \leftrightarrow a))$
$true$	$\left\{\begin{matrix}\bot, x,\\ y, xy\end{matrix}\right\}$	$\left\{\begin{matrix}xy, y,\\ x, \bot\end{matrix}\right\}$	$\langle ac, ab\rangle$	$\exists_{a,b,c}.((x \leftrightarrow a \wedge c) \wedge (y \leftrightarrow a \wedge b))$

Table 1: Representations of Boolean functions on $\{x, y\}$.

The class of definite Boolean functions is a complete lattice ordered by implication. The meet of two functions is their conjunction (Def is closed under conjunction) and the join is obtained by closing their disjunction under intersection of models (Def is not closed under disjunction).

In our representation the join operation is defined by pointwise union. For example, the joins of x and $x \leftrightarrow y$ and of x and y are obtained as: $\langle \bot, b\rangle \sqcup \langle a, a\rangle = \langle a, ab\rangle$ and $\langle \bot, b\rangle \sqcup \langle a, \bot\rangle = \langle a, b\rangle$ which correspond to $y \rightarrow x$ and $true$ respectively. The meet is defined in terms of ACI1 unification. For example the meet of $x \rightarrow y$ and $y \rightarrow x$ is obtained as the ACI1 unification of the corresponding tuples $\langle ab, a\rangle$ and $\langle c, cd\rangle$. The unification involves solving the system of ACI1 equations $\{ab = c, a = cd\}$ and applying the result on the given tuples. The result is $\langle a', a'\rangle$ which corresponds to the formula $x \leftrightarrow y$. In the sequel we formalize these definitions.

3 Preliminaries

Boolean Functions

Let $B = \{true, false\}$. A Boolean function on $V = \{x_1, \ldots, x_n\}$ is a function $\varphi : B^n \to B$. An *interpretation* $\mu : V \to B$ is an assignment of truth values to the variables in V. An interpretation μ is a *model* for φ, denoted $\mu \models \varphi$, if $\varphi(\mu(x_1), \ldots, \mu(x_n)) = true$. We write an interpretation as the set of variables which are assigned to the value *true*. The set of models of φ is thus viewed as a set of sets of variables defined by

$$[\![\varphi]\!]_V = \left\{ \{x \in V \mid \mu(x) = true\} \,\middle|\, \mu \models \varphi \right\}.$$

Much of the time we will omit the subscript V as it will be clear from the context.

Let φ be a Boolean function on V. We say that φ is *positive* if $V \in [\![\varphi]\!]$, that is, $\varphi(true, \ldots, true) = true$. We say that φ is *down-closed* if $[\![\varphi]\!]$ is closed under intersection. We denote by $\varphi{\downarrow}$ the strongest logical consequence of φ which is down-closed. In other words, $[\![\varphi{\downarrow}]\!] = [\![\varphi]\!]{\downarrow}$ is the smallest set that contains $[\![\varphi]\!]$ and is closed under intersection.

The class of positive Boolean functions ordered by logical consequence is a complete lattice denoted as Pos (sometimes we write Pos_V). An element of Pos is *definite* if it is down-closed. The definite Boolean functions form a sub-lattice of Pos denoted by Def (sometimes we write Def_V).

The *dual* of a Boolean function is the function that results when the roles of *false* and *true* are interchanged [21]. For any Boolean function φ we denote by $coneg(\varphi)$ the dual of the negation of φ (or, equivalently, the negation of its dual). In terms of the models of a function φ we have the following *pointwise complementation principle* [10] which states that to move from $[\![\varphi]\!] = \{m_1, \ldots, m_n\}$ to its dual negated counterpart, replace each element m_i by its complement:

$$[\![coneg(\varphi)]\!]_V = \{\, V \setminus m \mid m \in [\![\varphi]\!]\}.$$

ACI1 Equivalence and Tuples of Sets

We assume a denumerable set of variables $\mathcal{V}$ and an alphabet $\Sigma = \{\oplus, \bot\}$ consisting of a binary function symbol $\oplus$ and a constant symbol $\bot$. Sets of variables from $\mathcal{V}$ are represented as elements of the term algebra $T(\Sigma, \mathcal{V})$ modulo an equality theory consisting of the following axioms:

$$\begin{array}{ll}
(x \oplus y) \oplus z = x \oplus (y \oplus z) & (associativity) \\
x \oplus y = y \oplus x & (commutativity) \\
x \oplus x = x & (idempotence) \\
x \oplus \bot = x & (unit\ element)
\end{array}$$

The corresponding equivalence relation on terms is denoted $\approx_{aci1}$. This notion of equivalence suggests that set expressions can be viewed as flat sets of variables. For example, the terms $x_1 \oplus x_2 \oplus x_3$, $x_1 \oplus x_2 \oplus x_3 \oplus \bot$, and $x_1 \oplus x_2 \oplus x_3 \oplus x_2$ can each be viewed as representing the set $\{x_1, x_2, x_3\}$ of three variables. The set of n-tuples of set expressions is denoted $\mathsf{Tuple}_n(\Sigma, \mathcal{V})$. Set atoms are entities of the form $p(\bar{\tau})$ where p/n is an n-ary predicate symbol and $\bar{\tau}$ is an n-tuple of set expressions.

Set substitutions are substitutions which map variables of $\mathcal{V}$ to set expressions from $T(\Sigma, \mathcal{V})$. The application of a set substitution μ to a syntactic object τ is defined as usual by replacing occurrences of each variable x in τ by the set expression $\mu(x)$. The standard operations on set substitutions such as projection and composition are also defined just as for usual substitutions.

A preorder $\preceq_{aci1}$ on tuples of set expressions (and other syntactic objects) is defined as usual so that $\tau_1 \preceq_{aci1} \tau_2$ if there exists a set substitution μ such that $\tau_1 =_{aci1} \mu(\tau_2)$. In this case we say that τ_1 is less instantiated (modulo ACI1) than τ_2. This preorder induces a corresponding equivalence relation $\approx_{aci1}$ on tuples and a partial order $\preceq_{aci1}$ on the equivalence classes. For each natural n, $\mathsf{Tuple}_n = (\mathsf{Tuple}_n(\Sigma, \mathcal{V})/_{\approx_{aci1}}, \preceq_{aci1})$ is a complete lattice. The join is defined by

$$\langle \tau_1, \ldots, \tau_n \rangle \sqcup \langle \tau'_1, \ldots, \tau'_n \rangle = \langle (\tau_1 \oplus \tau'_1), \ldots, (\tau_n \oplus \tau'_n) \rangle$$

and the meet is defined through ACI1 unification [2]. An ACI1 unifier for two tuples τ_1 and τ_2 is a set substitution μ such that $\mu(\tau_1) =_{aci1} \mu(\tau_2)$.

Example 3.1 *Consider the tuples* $\bar{\tau}_1 = \langle a \oplus c, a \oplus b \rangle$ *and* $\bar{\tau}_2 = \langle a', b' \rangle$. *Their ACI1 equivalence is illustrated by the substitutions* $\mu_1 = \{a \mapsto \bot, b \mapsto b', c \mapsto a'\}$ *and* $\mu_2 = \{a' \mapsto a \oplus c, b' \mapsto a \oplus b\}$.

In our case, the underlying alphabet contains only the two (interpreted) function symbols $\{\oplus, \bot\}$. In this case the unification problem is classified as *elementary unification* [2]. The elementary unification problem is trivial (there always exists a unifier which binds all variables to $\bot$), the problem is unitary (there always exists a unique most general unifier) and the unification algorithm is efficient. The unique most general unifier for two set expressions (in our special case) τ and τ' is a substitution μ which maps variables to sets of variables as follows:

$$\mu(X) = \left\{ Z_{ab} \;\middle|\; \begin{array}{c} (a, b) \in vars(\tau) \times vars(\tau') \\ and \;\; X \in \{a, b\} \end{array} \right\}$$

where a fresh variable Z_{ab} from $\mathcal{V}$ is introduced for each pair (a, b).

Example 3.2 *Let* $\tau = w \oplus u$ *and* $\tau' = x \oplus y$. *The most general ACI1 unifier of* τ *and* τ' *is obtained as:*

$$\mu = \begin{Bmatrix} w \mapsto Z_{wx} \oplus Z_{wy}, & u \mapsto Z_{ux} \oplus Z_{uy}, \\ x \mapsto Z_{ux} \oplus Z_{wx}, & y \mapsto Z_{uy} \oplus Z_{wy} \end{Bmatrix}$$

4 Boolean Functions as Tuples of Sets

Let $V \subseteq \mathcal{V}$ be a finite set of variables. A set $M = \{m_1, \ldots, m_k\}$ of subsets of V can be represented as an incidence structure $S = (V, \mathcal{M}, \rho)$ where $\mathcal{M} \subseteq \mathcal{V}$ is a set of variables, disjoint from V, corresponding to the (names of the) sets in M and $\rho \subseteq V \times \mathcal{M}$ is a relation which specifies which elements from V belong to which sets from M. If $(x, m) \in \rho$ then we say that x and m are *incident*. For $m \in \mathcal{M}$ and $x \in V$ we denote the elements of m by $(m) = \{y \in V \mid (y, m) \in \rho\}$ and the sets containing x by $(x) = \{q \in \mathcal{M} \mid (x, q) \in \rho\}$. For $V = \{x_1, \ldots, x_n\}$, we denote the n-tuple $\langle (x_1), \ldots, (x_n) \rangle$ by $\mathcal{T}(M)$. Since the elements of $\mathcal{M}$ are interpreted as sets it follows that for any $m, q \in \mathcal{M}$, if $(m) = (q)$ then $m = q$.

Example 4.1 *Consider the set $M = \{\bot, x, y, xy\}$ over $V = \{x, y\}$. Its representation as an incidence structure is illustrated in Figure 1.*

$$
\begin{array}{lll}
(a) = \{x\} & (x) = \{a, b\} & \\
(b) = \{x, y\} & (y) = \{b, c\} & \\
(c) = \{y\} & & \\
(d) = \emptyset & \langle (x), (y) \rangle = \langle \{a, b\}, \{b, c\} \rangle &
\end{array}
$$

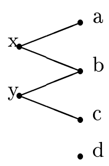

Figure 1: Incidence structure for $\{\bot, x, y, xy\}$.

Viewing a set of sets of variables M over $V = \{x_1, \ldots, x_n\}$ as an incidence structure $(V, \mathcal{M}, \rho)$ provides two alternative representations. The original set M is obtained as $\{(m) \mid m \in \mathcal{M}\}$. A dual representation is obtained as $\mathcal{T}(M) = \langle (x_1), \ldots, (x_n) \rangle$. Note that the number of symbols in the two representations is always the same. Note also that the dual representation is ambiguous concerning the presence of the empty set. Luckily, in our application the empty set will always be present and hence there will be no ambiguity. This is because $\varphi \in \mathsf{Pos}$ if and only if $[\![coneg(\varphi)]\!]$ contains the empty set.

In the remainder of this section we prove that for $V = \{x_1, \ldots, x_n\}$, Def_V is isomorphic to Tuple_n.

Definition 4.1 *Let $V = \{x_1, \ldots, x_n\}$. We define the mapping $iff : \mathsf{Tuple}_n \to \mathsf{Def}_V$ such that for $\bar{\tau} = \langle \tau_1, \ldots, \tau_n \rangle \in \mathsf{Tuple}_n$:*

$$iff(\bar{\tau}) = \exists_{vars(\bar{\tau})}. \bigwedge_i (x_i \leftrightarrow \wedge \tau_i)$$

where $vars(\bar{\tau})$ denotes the set of variables in $\bar{\tau}$ and $\wedge \tau_i$ is the conjunction of the variables in the i^{th} argument τ_i of $\bar{\tau}$. Note that $iff(\bar{\tau}) \in \mathsf{Def}$ because Def is closed under existential quantification and $x_i \leftrightarrow (a_1 \wedge \cdots \wedge a_n)$ is equivalent to $[(a_1 \wedge \cdots \wedge a_n) \to x_i] \wedge [(x_i \to a_1) \wedge \cdots \wedge (x_i \to a_n)]$ which is in propositional definite form.

Example 4.2 *Consider $\bar{\tau} = \langle a \oplus b, b \rangle$. We have $iff(\bar{\tau}) = \exists_{a,b}.(x_1 \leftrightarrow a \wedge b) \wedge (x_2 \leftrightarrow b)$ which is equivalent to $x_1 \to x_2$.*

Lemma 4.1 *Let $\varphi \in \mathsf{Pos}$ and $\bar{\tau} = \mathcal{T}([\![coneg(\varphi)]\!])$ then $iff(\bar{\tau}) = \varphi\!\downarrow$*

Lemma 4.2 *Let $\tau_1, \tau_2 \in \mathsf{Tuple}_n$ then: $\tau_1 \leq_{aci1} \tau_2 \Leftrightarrow (iff(\tau_1) \to iff(\tau_2))$.*

Theorem 4.1 *Let $V = \{x_1, \ldots, x_n\}$ then Tuple_n and Def_V are isomorphic domains.*
PROOF. *This follows from Definiton 4.1, Lemma 4.1 and Lemma 4.2* $\square$

5 Groundness Analysis using Tuple_n

The specification of a program analysis within the framework of abstract interpretation requires: (1) a concrete semantics, (2) an abstract domain, and (3) abstract operations. Our abstract domain is Tuple_n and is isomorphic to Def. This section focuses on the abstract operations required to support groundness analysess as determined by the standard choices of concrete semantics. In addition, we suggest two widening techniques which can be applied to improve the scalibility properties of the analysis.

Representation and Abstract Operations

We adopt a non-ground representation. The sets of variables in a tuple are represented as lists of (Prolog) variables. In this way we benefit from the underlying renaming mechanism of Prolog. For example, the Def object $p(x_1, x_2) \leftarrow (x_1 \rightarrow x_2)$ is represented by $p([A, B], [A])$.

The isomorphism between Def and Tuple establishes the formal correspondence between the domain elements and operations (join and meet). The complexity of the join operator is linear in the number of the arguments (pointwise union). The complexity of the meet operator (ACI1 unification) is quadratic in the size of the atoms being unified. Projection is also straightforward to define (it is the same as the standard operation on atoms).

There is one additional operation that we must consider. Given two elements from Tuple_n (which are representatives of equivalence classes), we need to determine if they are ACI1 equivalent. To this end we introduce a *normal form* representation and hence an operation which given a tuple of sets computes its normal form.

Figure 2 illustrates the definition of the normal form operation which is more easily described through Def. We have a tuple of sets of variables $\bar{\tau}$ (on the top right side of the figure) with $\mathit{iff}(\bar{\tau}) = \varphi$ (on the top left). We take as φ' (bottom left) the conjunction of all of the Boolean functions $\psi \in \mathsf{Pos}$ such that $\psi{\downarrow} = \varphi$. Stated alternatively: $[\![\varphi]\!]'$ is the meet irreducible (m.i.) subset of $[\![\varphi]\!]$. Namely we remove from $[\![\varphi]\!]$ any model obtained as the intersection of other models. The existence and uniqueness of φ' is justified by Proposition 5.1 (below) . Our normal form is the tuple (on the bottom right side of the figure) $\bar{\tau}' = \mathcal{T}(\mathit{coneg}(\varphi'))$.

$$\begin{tikzcd}
\varphi \arrow[d, "\text{m.i.}"'] & & \bar{\tau} \arrow[ll, "\mathit{iff}(\cdot)"'] \arrow[d, Leftrightarrow] \\
\varphi' \arrow[rr, "\mathcal{T}(\mathit{coneg}(\cdot))"] & & \bar{\tau}'
\end{tikzcd}$$

Figure 2: Normal form representation in Tuple

Proposition 5.1 *Let $\varphi_1, \varphi_2 \in \mathsf{Pos}$ and $\varphi_1{\downarrow} = \varphi_2{\downarrow}$ then $(\varphi_1 \wedge \varphi_2){\downarrow} = \varphi_1{\downarrow} = \varphi_2{\downarrow}$.*

In the implementation, the normal form is computed directly on the tuple representation. Whereas in Def we removed sets obtained as the intersection of others, for tuples (which encode the *coneg* of the formula) we should remove variables that correspond to models obtained as the union of others. The operation is described viewing the n-tuple in terms of the incidence structure it represents. We demonstrate the computation of the normal form of the tuple

$$\langle \underbrace{abce}_{x}, \underbrace{acd}_{y}, \underbrace{ae}_{z} \rangle$$

which represents the *coneg* models

$$\left\{ \begin{array}{l} a = \{x, y, z\}, b = \{x\}, c = \{x, y\}, \\ d = \{y\}, e = \{z, x\}, f = \bot \end{array} \right\}.$$

The corresponding incidence structure is illustrated in Figure 3 (on the left). We

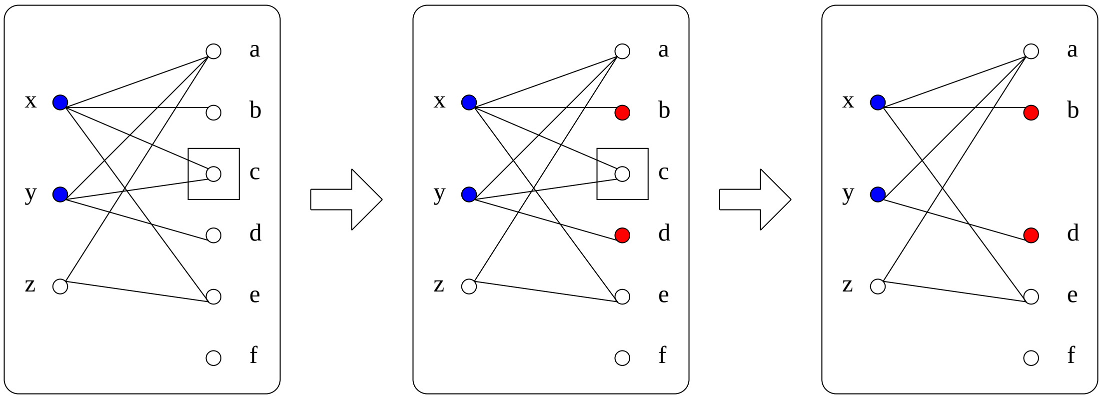

Figure 3: Computing a normal form: eliminating c

should eliminate a (because it is obtained as the union of c and d) and c (because it is obtained as the union of d and b).

In the following we refer to the points on the left of the incidence structure (x,y,z) as *variables* and to those on the right (a,b,c,d,e) as *models*. The algorithm considers each model m to determine if can be obtained as the union of other models as follows:

1. color the variables connected to m;
2. color the models, except m, which are connected only to colored variables;
3. if each of the colored variables is connected to at least one colored model then remove m.

Figure 3 depicts the process of eliminating the model c in our example. First the variables x and y are colored (as these are connected to c) and then the models b and d are colored (as these are connected only to colored variables). The colored variables are each connected to at least one colored model, so c is removed. Repeating this opeartion for all models gives the normal form $\langle be, d, de \rangle$.

The complexity of the algorithm is O(k*n), where k and n are the number of vertices and edges in the incidence structure.

Widening the tuples

The size of a (normal form) representation of a tuple of sets is potentially exponential in the number of sets. This is important because the operations for meet and for computing a normal form (as implemented) are quadradic in the size of their inputs. As an illustration, consider the n-ary Boolean function whose set of models consist of the empty set together with the $\binom{n}{n/2}$ sets with $\frac{n}{2}$ variables. This set of models is meet irriducible and of exponential size. Hence so is the normal form of its dual representation as a tuple of sets.

In addition to the potential size problem, we also know from [16] that the number of iterations in a Def analysis has a worst case of at least $2^{\sqrt{n}}$. To guarantee a robust analysis we introduce two possible widening operations on our tuple representation.

Restricting the size of sets: With this approach we restrict the size, k, of the sets in a (normalized) tuple. With $k = 1$ we can express groundness information and equivalences between variables. For example, as in $p([A], [\,], [A], [B], [\,])$ which corresponds in Def to $p(x_1, x_2, x_3, x_4, x_5) \leftarrow (x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$. With $k = 2$ we can express simple dependencies between variables as in $p([A, B], [A])$ which corresponds in Def to $p(x_1, x_2) \ \leftarrow \ (x_1 \rightarrow x_2)$ and as in $append([A], [B], [A, B])$ which corresponds in Def to $append(x_1, x_2, x_3) \ \leftarrow \ ((x_1 \wedge x_2) \leftrightarrow x_3)$.

The widening operation for a given value of k should take a tuple $\bar{\tau}$ which might contain sets which are larger than k and provide a possibly more general tuple $\bar{\tau}'$ in which the sets have no more than k elements.

A naive solution is to "drop" large sets (those with more than k elements). For example, the atom $\pi = \ p([A, B, C, D], [A, B, C], [A, B], [A])$ could be widened with $k = 2$ to $p([\_], [\_], [A, B], [A])$. Less naive is to first rename the variables in the large sets and recompute the normal form. As a consequence, we maintain (some of) the dependencies between the variables in the large sets. For the above example, π is widened in this way with $k = 2$ to $p([X, Y], [X], [A, B], [A])$.

It is interesting to note that widening with $k = 1$ results in an analysis over the domain $EPos$ described in [17] which captures groundness information and equivalences between pairs of variables. This simple domain was shown in [17] to be suitable for goal dependent groundness analysis of logic programs. The reader familiar with that work will note that the most precise analysis described on $EPos$ lacked in efficiency due to the non-determinism of the abstract operations (the `iff/2` atoms). With our approach, taking $k = 1$ gives the same precision as in that analysis - but due to the use of ACI1 unification the abstract operations are now determinstic. This results in a fast analysis.

Restricting the size of unifications: An alternative approach to avoid generating large sets in tuples is to widen the operations that generate complex dependencies. Given a sequence of ACI1 unifications that need to be solved (for example corresponding to unifying the the atoms of a clause body with corresponding current approximations for these atoms) we can obtain a more general solution by "dropping" unifications between sets which are too large. Recall that the unification between sets of n and m variables generates $n * m$ fresh variables. Hence if n and

m are large then this operation introducing a very larger number of fresh variables. Subsequently, the normalisation algorithm can be extremely costly.

With this approach we restrict the size $u = min(m, n)$ of a unifcation algorithm that we solve. For example, with $u = 1$ we drop any unification for which both m and n are larger than 1.

6 Experimentation

We have implemented a prototype analyser based on the Tuple domain isomorphic to Def for groundness analysis. The implementation is based on a simple bottom-up meta-interpreter enhanced by ACI1 unification. For goal-independent analyses we use a seminaive interpreter otimized for strongly connected components. For goal-dependent analyses we use an interpreter which applies induced magic-sets [5] and eager evaluation [22]. This is basically the same analysis engine used in [19] and [18]. The implementation effort itself involved a small effort given the analysers described in [8] and in [7]. The main difference is in the simplifications made in the unification algorithm and in the normal form algorithm.

We have experimented with the set of 46 Prolog programs from the benchmark suit used in [18]. For goal dependent analyses the analyser exhibits timings similar to those described in [19] with the difference that the analyser *"times out"* on *fewer* of the benchmark programs.

For goal dependent analyses: Widening to restrict the size of variable sets to $k = 2$ gives no loss of precision for all of the benchmark programs except one (for the *aqua* compiler benchmark we loose 33 out of the 1285 ground arguments); Widening to restrict the size of unifications taking $u = 1$, gives no loss of precision for all of the benchmark programs. These are extreme widennings which lead to very fast analyses.

For goal independent analyses, it is interesting to note that although Def is known not to be a condensing domain, the amount of groundness obtained when applying a goal independent analysis is precisely the same as with the goal dependent analysis for the set of benchmarks chosen. However, for goal independent analyses, extreme widenings such as $k = 2$ and $u = 1$ are not realistic with regards to the loss of precision.

7 Conclusion

The study of the relationships between Pos, Sharing and Def has been the topic of many research papers in the last 10 years. Analyses using Def were defined early on in the work of Dart [14] and an evaluation of various representations for Def can be found in [1]. However, the use of BDD's for Pos analysis is more popular. Due to the difficulty in widening BDD representations, it is a recent trend to look for weaker domains which support analyses that are fast and do not exhibit (or can be widened so as not to exhibit) the underlying worst case complexity for the occasional hard example. Recent results described in [19] and [18] indicate that Def is a promising candidate for the groundness analysis of (constraint) logic programs.

We provide a new representation for Def based on n-tuples of sets of variables as proposed in [8] for representing set Sharing. Our main result states that the quotient of Sharing to Def is precisely the quotient of our n-tuples representation with respect to ACI1 equivalence. From the practical point of view this provides the basis to implement program analysis over Def using the classic and well-studied ACI1 unification algorithms. For the special case where the only constant symbol is the unit element this is efficient and simple to implement.

As a curiosity, it is interesting to note that formalizing tuples of sets of variables (and sets of sets of variables) as incidence structures sheds new light also on the results of [8]. In particular, we note that while the notion of equivalence adopted in [8] seems ad-hoc and specialised for the isomorphism with *Sharing*, it is a standard notion of equivalence on incidence structures as used in finite geometry [15].

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[2] F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–126. Oxford Science Publications, 1994.

[3] Roberto Bagnara and P. Schachte. Factorizing equivalent variable pairs in ROBDD-based implementations of *Pos*. In A. M. Haeberer, editor, *Proceedings of the "Seventh International Conference on Algebraic Methodology and Software Technology (AMAST'98)"*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485, Amazonia, Brazil, 1999. Springer-Verlag, Berlin.

[4] R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[5] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):354–370, 1999.

[6] M. Codish. Worst-case groundness analysis using positive Boolean functions. *Journal of Logic Programming*, 41(1):125–128, 1999.

[7] M. Codish and V. Lagoon. Type dependencies for logic programs using aci-unification. *Theoretical Computer Science*, 2000.

[8] M. Codish, V. Lagoon, and Bueno F. An algebraic approach to sharing analysis of logic programs. *Journal of Logic Programming*, 41(2):110–149, 2000.

[9] M. Codish and H. Søndergaard. The Boolean logic of set sharing analysis. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 89–101, Berlin, 1998. Springer.

[10] M. Codish, H. Søndergaard, and P. J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

[11] A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation. *Theoretical Computer Science*, 202(1–2):163–192, 1998.

[12] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Evaluation of the domain Prop. *Journal of Logic Programming*, 23(3):237–278, 1995.

[13] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. Number 631 in Lecture Notes in Computer Science, pages 269–295, Leuven, Belgium, 1992. Springer-Verlag, Berlin.

[14] P. W. Dart. On derived dependencies and connected databases. 11(1 & 2):163–188, July 1991.

[15] P. Dembowski. *Finite Geometries.* Springer-Verlag, 1968.

[16] S. Genaim, J.M. Howe, and M. Codish. Worst-case groundness analysis using definite boolean functions, September 2000. http://www.cs.bgu.ac.il/~mcodish/Papers/ppapers.html.

[17] A. Heaton, M. Abo-Zaed, M. Codish, and A. King. A Simple Polynomial Groundness Analysis for Logic Programs. *Journal of Logic Programming*, 45:143–156, September 2000.

[18] J. M. Howe and A. King. Implementing groundness analysis with definite Boolean functions. In G. Smolka, editor, *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2000.

[19] A. King, J. G. Smaus, and P. Hill. Quotienting share for dependency analysis. In Doaitse Swierstra, editor, *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, April 1999.

[20] K. Marriott and H. Søndergaard. A tutorial on abstract interpretation of logic programs. Notes distributed at the North American Conference on Logic Programming, 1989.

[21] E. Schröder. *Der Operationskreis des Logikkalküls.* B. G. Teubner, Leibzig, Germany, 1877.

[22] J. Wunderwald. Memoing evaluation by source-to-source transformation. In Maurizio Proietti, editor, *Proceedings of the Fifth International Workshop on Logic Program Synthesis and Transformation*, volume 1048 of *LNCS*, pages 17–32. Springer, 1995.